

Lins: Reducing Communication Overhead of ZeRO for Efficient LLM Training

Qiaoling Chen^{1,2,3}, Qinghao Hu^{1,2,3}, Guoteng Wang³, Yingtong Xiong³, Ting Huang⁴, Xun Chen⁴
Yang Gao³, Hang Yan³, Yonggang Wen², Tianwei Zhang², Peng Sun^{3,4}

¹S-Lab, NTU ²Nanyang Technological University ³Shanghai AI Laboratory ⁴SenseTime

Email: {qiaoling.chen, qinghao.hu, ygwen, tianwei.zhang}@ntu.edu.sg

{wangguoteng, xiongyingtong, gaoyang, yanhang}@pjlab.org.cn, {huangting3, chenxun, sunpeng1}@sensetime.com

Abstract—Training large language models (LLMs) encounters challenges in GPU memory consumption due to the high memory requirements of model states. The widely used Zero Redundancy Optimizer (ZeRO) addresses this issue through strategic sharding but introduces communication challenges at scale. To tackle this problem, we propose Lins, a system designed to optimize ZeRO for scalable LLM training. Lins incorporates three flexible sharding strategies: *Full-Replica*, *Full-Sharding*, and *Partial-Sharding*, and allows each component within the model states (Parameters, Gradients, Optimizer States) to independently choose a sharding strategy as well as the device mesh. We conduct a thorough analysis of communication costs, formulating an optimization problem to discover the optimal sharding strategy. Evaluations demonstrate up to 52% Model FLOPs Utilization (MFU) when training the LLaMA-based model on 1024 GPUs, resulting in a 1.56 times improvement in training throughput compared to newly proposed systems like MiCS and ZeRO++.

I. INTRODUCTION

Large Language Models (LLMs) have demonstrated exceptional performance in various tasks, with the relationship between model size and performance often following a power-law relationship. Despite the prevailing trend of training giant models like GPT-3 with 175 billion parameters, recent studies indicate that optimal performance may be achieved with smaller models trained on larger datasets [1]. Emerging LLMs like LLaMA [2], featuring 7 billion to 30 billion parameters.

Training LLMs significantly demands on GPU memory, primarily due to the substantial memory consumption of model states, encompassing parameters (P), gradients (G), and optimizer states (OS). Additional memory is allocated for activations and temporary buffers. For instance, when training LLaMA-7B, a substantial 112GB of memory is required for model states, surpassing the capacity of an 80GB NVIDIA A100 GPU. To address this challenge, ZeRO, implemented in Deepspeed [3] and PyTorch FSDP [4], introduces a sharding strategy to alleviate redundant memory allocations. ZeRO-1 distributes optimizer states across GPUs, ZeRO-2 further shards gradients and ZeRO-3 extends this approach to parameters, gradients, and optimizer states. This strategic sharding optimizes memory usage, enabling efficient training of large models within GPU constraints. ZeRO could work in cooperation with 3D parallelism [5] and has become widely adopted in distributed LLMs training.

ZeRO heavily relies on collective communication for effective model states management, introducing challenges in large-

scale LLM training due to the substantial transmission cost. In our experiments, training a LLaMA-7B model on 8 GPUs using ZeRO-1 achieves a model FLOPs utilization (MFU) of 63%, but scaling to 1024 GPUs with the same batch size results in a significant performance reduction, with the MFU dropping to 36%. The costly communication of ZeRO can be attributed to three primary factors: 1) a significant bandwidth discrepancy between inter-node and intra-node networks, 2) an increase in collective communication latency as the communication scale grows, and 3) the use of a small micro-batch size per GPU on numerous GPUs, exacerbating the compute-to-communication ratio imbalance.

Several approaches have been proposed to reduce the communication overhead of ZeRO with improved memory utilization. ZeRO++ [6] achieves this by maintaining a secondary parameters shard within small subgroups, effectively reducing communication latency when collecting them. MiCS [7] shards all model states components within subgroups and replicates them across subgroups, reducing communication scale and consequently reducing communication latency, leading to enhanced training performance. Despite these efforts, when scaling LLM training to a large extent, ZeRO++ and MiCS exhibit suboptimal speedup ratios due to the inflexible model states sharding mechanism results in suboptimal communication costs. This limitation is evident in the case of MiCS, where scaling LLaMA-7B training from 8 GPUs to 1024 GPUs leads to a significant decrease in model training performance, even falling below the efficiency of ZeRO-1.

We propose Lins for reducing the communication overhead of ZeRO for training LLMs at scale. To achieve this goal, Lins incorporates three flexible sharding strategies—*Full-Replica*, *Full-Sharding*, and *Partial-Sharding*—allowing each component within the model states (P , G , OS) to independently choose a sharding strategy. The introduced sharding factors ($s_p^0 \times s_p^1, s_g^0 \times s_g^1, s_{os}^0 \times s_{os}^1$) control the number of GPUs and the device mesh over which the tensors are sharded. Given this flexibility, we analyze the memory consumption and communication costs for each sharding dimension. Then, we formulate an optimization problem aimed at discovering optimal sharding factors that minimize communication costs while adhering to the constraint of GPU memory capacity. Lins implements an execution engine tailored for training LLMs, incorporating these flexible sharding factors to achieve

TABLE I
NOTATIONS USED IN THIS WORK.

Notation	Meaning
D	Memory consumption of a GPU.
T	Time consumption.
Φ	Model parameters count during training.
N	Total number of GPU nodes used for training.
R	Number of GPUs per computational node.
B	Micro-batch size (sequences per micro-batch).
M	Number of micro-batches.
L	Number of layers of the model.
K	Number of modules within a layer of the model.
s_{dp}, s_{tp}, s_{pp}	Size of data, tensor and pipeline parallelism.
s_p, s_g, s_{os}	Sharding factors of parameters, gradients and model states.

optimized communication efficiency during training.

Extensive evaluations show a significant system performance of `Lins` on training LLaMA-based models. On 1024 Nvidia Ampere GPUs, the MFU of `Lins` is 51%, 52%, and 42% on LLaMA-7B, LLaMA-13B, and LLaMA-30B training. In comparison, MiCS demonstrates lower MFU values at 35%, 33%, and 29% for the same models, ZeRO++ shows the least MFU among the three, with MFU rates at merely 4%, 6%, and 5% for the 7B, 13B, and 30B models, respectively. Compared to MiCS and ZeRO++, `Lins` improves the training throughput by a factor of 1.4 – 12.7 on 1024 GPUs for training LLaMA-based models. `Lins`¹ has been used for training InternLM on thousands of GPUs. Our efforts also encompass an exhaustive study characterizing a six-month development workload trace of LLM collected from our GPU datacenter [8].

II. BACKGROUND

We provide a brief introduction to the essential background of LLM training and the associated challenges to improve performance. Table I gives notations used in this work.

A. LLM Architecture

LLMs like GPT-3 [9] and LLaMA [2] widely adopt the Transformer [10] architecture with multiple layers. Each Transformer layer comprises a list of modules, such as linear, multi-head-attention (MHA), and norm modules. The input and output dimensions for each Transformer layer are denoted as $B \times S \times H$, where B represents the micro-batch size, S indicates the sequence length, and H is the hidden dimension. The relationship between the model size of LLMs and their performance is typically governed by a power-law relationship. While there has been a trend to train giant models like GPT-3 with 175B parameters, existing studies suggest that optimal model performance may be attained with smaller models trained on larger datasets [1]. As illustrated in Table II, recently introduced LLMs like LLaMA and InternLM typically feature 7B to 30B parameters.

TABLE II
POPULAR LLMs AND THEIR PARAMETERS COUNT.

Model	# Parameters	Model	# Parameters
GPT-3	175B	BLOOM	175B
LLaMA	7B, 13B, 33B, 65B	Mistral	7B
LLaMA2	7B, 13B, 70B	InternLM2	7B, 20B
Cerebras-GPT	1.3B, 2.7B, 6.7B, 13B	Baichuan2	7B, 13B

B. Distributed LLM Training

Efficiently training LLMs at scale in GPU clusters involves utilizing 3D parallelism. Data Parallelism (DP) divides input data into chunks, distributing them across GPUs, where each GPU independently computes gradients, later synchronized through `AllReduce` communication [11]. Tensor Parallelism (TP) distributes parameters across GPUs along specific dimensions for parallel training. Megatron-LM employs TP to partition linear layers along the row or column dimension, integrating collective communication operations for consistent results [5]. Pipeline Parallelism (PP) evenly divides a model’s Transformer layers into multiple stages, distributing them across GPUs. A scheduler splits an input batch into micro-batches, alternating between forward and backward computations [12] [13]. Two consecutive pipeline stages exchange intermediate data through point-to-point communication.

C. ZeRO

Training LLMs results in significant memory consumption, largely due to the occupation of GPU memory by model states, which comprise tensors containing parameters (P), gradients (G), and optimizer states (OS). The remaining memory is allocated to activations and temporary buffers. In the context of a model with Φ parameters, employing mixed precision training alongside the Adam optimizer [14], it necessitates 2Φ , 2Φ , and 12Φ bytes of GPU memory for P , G and OS , respectively. As an illustrative example, the LLaMA-7B model requires 112GB of memory for its model states, exceeding the memory capacity of an NVIDIA A100 GPU (80GB). As shown in Figure 1, ZeRO reduces redundant memory usage for model training by sharding model states [15].

ZeRO-1 splits optimizer states across GPUs ($s_{os} > 1$). In the training phase, each GPU independently computes gradients through forward and backward computations, which are then synchronized across s_{dp} GPUs using `AllReduce`. Each GPU updates specific portions of the model parameters. The most recent model parameters for a GPU are gathered from other GPUs using the `AllGather` operation. ZeRO-2 extends this approach by further sharding gradients across GPUs ($s_g = s_{os} > 1$). Each GPU retains only the gradients corresponding to its optimizer states segment after the reduction operation.

ZeRO-3, also implemented in FSDP [4], employs the sharding strategy on model parameters, gradients, and optimizer states ($s_p = s_g = s_{os} > 1$). Before each forward and backward computation, individual GPUs execute the `AllGather` operation to assemble the complete set of model parameters

¹Please visit <https://github.com/InternLM/InternEvo> to access the system.

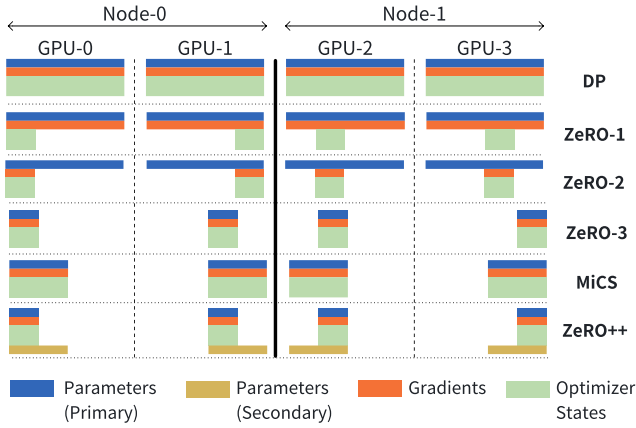


Fig. 1. Overview of GPU memory allocation for model states with different strategies. ZeRO-1 and ZeRO-3 significantly reduce memory consumption for model states compared to standard data parallelism. MiCS and ZeRO++ are proposed to mitigate communication overhead, particularly cross-node communication time, in comparison to the ZeRO approach.

and subsequently discard them post-computation. The synchronization of gradients across GPUs is achieved through Reduce-Scatter. Each GPU updates its corresponding shard of model parameters using the maintained optimizer states and gradients at the end of each step.

III. CHALLENGES AND MOTIVATION

ZeRO has gained extensive adoption across various training frameworks, such as DeepSpeed [3], FSDP [4], and ColossalAI [16], owing to its user-friendly interface and scalability across hundreds of GPUs. Despite leveraging high-bandwidth RDMA networks, challenges emerge in the form of poor Quality of Service (QoS) during distributed LLM training on large-scale GPU clusters. This can be mainly attributed to significant communication overhead.

A. High Communication Overhead

ZeRO necessitates extensive usage of collective communication for managing parameters and gradients. The transmission cost across large-scale clusters presents a challenge, as it cannot be easily mitigated through computation-communication overlapping. When training a LLaMA-7B model on 8 GPUs using ZeRO-1, the model FLOPs utilization (MFU) attains 63% in our test-bed. Scaling the training to 1024 GPUs with the same batch size results in a notable performance reduction, with the MFU dropping to 36%. Similarly, scaling LLaMA-13B training from 8 GPUs to 1024 GPUs with ZeRO-3 leads to a substantial MFU reduction from 47% to 4%.

Three main factors contribute to the costly communications for large-scale LLM training with ZeRO. Firstly, there exists a notable discrepancy between inter-node network bandwidth and intra-node NVLINK bandwidth. High-performance DGX-A100 nodes offer 600GB/s intra-node bidirectional bandwidth per GPU and provide 400GB/s inter-node bidirectional bandwidth per node. The bandwidth ratio between intra-node and inter-node measures at 2 in our test-bed. Secondly, the latency

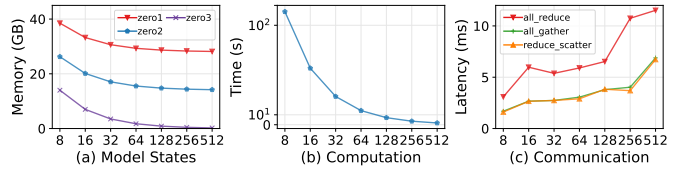


Fig. 2. Micro-benchmark of training LLaMA-7B across a scale of GPUs, ranging from 8 to 512, while maintaining a global batch size of 4M tokens. The micro-batch size B is consistently set to 1 in all tests. Panel (a) illustrates the GPU memory consumption of model states. Panel (b) depicts the time taken for forward and backward computations. Panel (c) presents the latency of three communication operations with a fixed message size of 256MB.

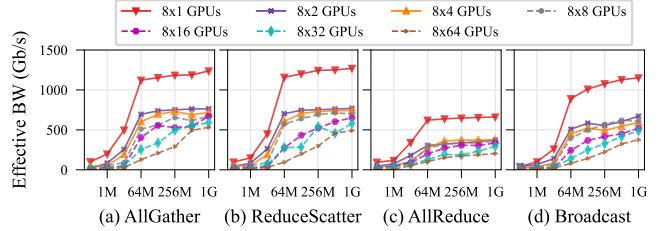


Fig. 3. Performance evaluation of collective communication operations using NCCL. The assessment is conducted with varying message sizes (in bytes). GPU nodes are linked using 4 Mellanox Infiniband HDR NICs (800 Gbps bandwidth in total). The notation $8 \times A$ GPUs indicates that the tests were conducted on A nodes, with each node housing 8 NVIDIA Ampere GPUs (A800) connected by NVLINK.

of collective communication operations demonstrates a positive correlation with communication scale [17] [18] [19] and illustrated in Figure 2(c). Figure 3 further illustrates a reduction in the effective bandwidth of communication operations utilized by ZeRO, scaling from 8 GPUs to 512 GPUs. Thirdly, the global batch size limitation for convergence efficiency imposes the use of a very small batch size per GPU when training on numerous GPUs. As depicted in 2 (b), the computation time of the LLaMA-7B model training linearly reduces from 8 GPUs to 512 GPUs while maintaining a consistent 4M batch size. This reduction adversely affects the compute-to-communication ratio, leading to a communication bottleneck.

B. Trade-off between Communication and Memory

A trade-off exists between memory utilization and communication cost in distributed LLM training. Initially, the communication cost can be effectively reduced by diminishing the communication scale. This involves limiting communications to a smaller group of GPUs, potentially within the same node, which mitigates the overall communication cost. In addition, as depicted in Figure 2 (a), scaling ZeRO to a large scale does not yield substantial memory savings compared to a smaller size. Consequently, various approaches have been proposed to reduce communication overhead with higher memory usage.

ZeRO++ [6] keeps a secondary shard of parameters while sharding other model states across the cluster ($s_p = s_g = s_{os} = s_{dp}$), as shown in Figure 1. In the forward phase, it collects parameters across all GPUs and maintains a secondary shard of parameters within a small subgroup of GPUs, potentially

within the same node. During the backward phase, it collects parameters from this secondary shard. Additionally, ZeRO++ uses quantization to compress parameters and gradients, effectively reducing inter-node communication size. Note that we would not enable configurations related to the quantization of ZeRO++ to ensure consistent model quality.

MiCS [7] and FSDP [4] facilitate the sharding of model states within a subgroup and replicate them across subgroups ($s_p = s_g = s_{os} < s_{dp}$), as shown in Figure 1. These approaches employ AllGather to collect parameters within a subgroup for both forward and backward computation and synchronize gradients across the cluster using ReduceScatter. Consequently, MiCS and FSDP contribute to improved training performance by effectively reducing the communication scale. It is crucial to configure an appropriate subgroup size to prevent Out-Of-Memory (OOM) errors.

C. Motivation

Despite efforts to reduce communication costs, ZeRO++ and MiCS still exhibit poor speedup ratios when scaling LLM training to a large scale. This is attributed to their inflexible model states sharding mechanism, requiring $s_p = s_g = s_{os} \leq s_{dp}$ in all cases. Such a configuration may not be optimal for training LLMs with diverse model sizes and hyper-parameters. For instance, when scaling LLaMA-7B training from 8 GPUs to 1024 GPUs with MiCS, MFU decreases from 50% to 35% in our test-bed. In this scenario, MiCS even exhibits lower performance than ZeRO-1, highlighting the drawbacks of the inflexible model states sharding mechanism.

In this study, the three components of model states (P , G , OS) are sharded into independent subgroups and replicated across these subgroups, following the condition $s_p \leq s_{dp}$, $s_g \leq s_{dp}$, $s_{os} \leq s_{dp}$. This flexibility allows us to fine-tune the trade-off between communication and GPU memory by configuring s_p , s_g , s_{os} . By doing so, we may achieve minimal communication cost for distributed LLM training through individualized configuration of the communication scale on P , G , and OS , while respecting GPU memory constraints.

Taking LLaMA-7B as an illustrative example, we adopt the configuration of $s_p = s_g = 1$, $s_{os} = 8$ for training. In this setting, each GPU retains a complete copy of parameters and gradients, while each node stores a full copy of optimization states. During training, gradients are synchronized across clusters using AllReduce, and each GPU obtains the latest parameters within the same node through AllGather at the end of each step. In our test-bed, scaling LLaMA-7B training from 8 GPUs to 1024 GPUs with this configuration results in an acceptable MFU reduction from 64% to 51%.

IV. MODEL STATES SHARDING AND ANALYSIS

In this section, we assume that there is no tensor parallelism or pipeline parallelism during the training, which implies that $s_{tp} = s_{pp} = 1$. This simplification allows us to focus on the impact of the discussed sharding strategies on communication and memory aspects.

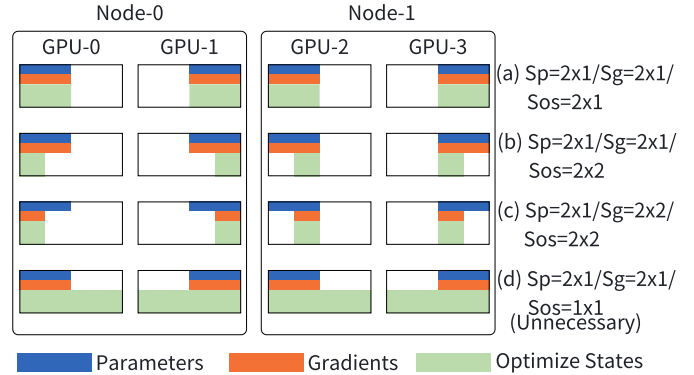


Fig. 4. Optimizing model states sharding through the dependency rule. In this instance, when $s_p = s_g = 2$, there's no need to set $s_{os} = 1$ as it would store redundant optimized states, incurring additional communication costs.

A. Performance Model of Collective Communication

The $\alpha - \beta$ cost model [20] is widely employed to characterize the performance of collective communication [?]. Taking the example of a ring-based AllReduce on p GPUs, where the input size is v , and the physical bandwidth between two GPUs is w , the input is evenly split into p chunks. In the first stage, each chunk undergoes $p - 1$ rounds of reduction to each GPU, constituting a ReduceScatter operation with a time complexity of $t_{rs} = (p - 1)(\alpha + \frac{v}{w \times p})$, where α denotes the latency per transmission. Then, each reduced chunk at every GPU is broadcast to other GPUs, constituting an AllGather operation with the same time complexity as ReduceScatter. The overall time complexity of the ring-based AllReduce is given by $t_{ar} = 2(p - 1)(\alpha + \frac{v}{w \times p})$.

However, predicting collective communication time with high accuracy using the $\alpha - \beta$ cost model is challenging in certain scenarios. First, in addition to the ring algorithm, NCCL introduces new communication algorithms like Tree [21], Collnet, CollnetDirect, and CollnetChain. Consequently, a single cost model struggles to formulate the communication time for all these algorithms. Second, In-Network Aggregation solutions are widely implemented in production GPU clusters. These solutions offload AllReduce onto network switches to accelerate and scale distributed training [22] [23].

In this work, we adopt a straightforward yet effective profiling-based approach to model the performance of collective communication. Specifically, we utilize

$$t(o, v, p^0 \times p^1) = v/w(o, v, p^0 \times p^1)$$

to evaluate the time consumption of a collective communication operator (o) with a given data size (v) and a specified participant GPU device mesh ($p^0 \times p^1$, where p^0 denotes the number of GPUs in a node, and p^1 is the number of nodes). Here, $w(o, v, p^0 \times p^1)$ represents the effective bandwidth obtained through performance profiling on the target GPU cluster in advance, as illustrated in Figure 3. In cases where v is not profiled, we employ the interpolation method to predict the effective bandwidth and communication time. This work focuses on four key collective commu-

nication operations: AllReduce (AR), AllGather (AG), ReduceScatter (RS) and Broadcast (BC).

B. Flexible Model States Sharding with Dependency Rule

We adopt three sharding strategies, namely *Full-Replica*, *Full-Sharding*, and *Partial-Sharding*, and provide the flexibility for each of the three components within the model states (P , G , and OS) to independently select a sharding strategy. To encapsulate these strategies, we introduce sharding factors $s_p = s_p^0 \times s_p^1$, $s_g = s_g^0 \times s_g^1$ and $s_{os} = s_{os}^0 \times s_{os}^1$, representing the number of GPUs and the device mesh over which the tensors of P , G , and OS are sharded, respectively. Setting the factor to 1 implies full replication of the tensor, simplifying to vanilla data parallelism if all components (P , G , and OS) choose the *Full-Replica* strategy. Conversely, setting the factor equal to the DP size results in complete tensor sharding, with each GPU holding $1/s_{dp}$ of the tensor. For instance, in ZeRO-3, all components (P , G , and OS) choose the *Full-Sharding* strategy. *Partial-Sharding* emerges when the factor falls between 1 and s_{dp} , indicating tensor sharding across a subgroup of GPUs and replication across subgroups.

A dependency rule is crucial when flexibly sharding the model states to avoid unnecessary data movement and storage. Throughout the training, the framework employs local parameters for gradient computation and synchronized gradients for updating local optimizer states. If a GPU oversees extra gradients or optimizer states unrelated to its local parameters, launching additional communication becomes necessary. This incurs significant and avoidable expenses. Figure 4 illustrates an instance of this scenario, highlighting the impact when setting $s_p = s_g = 2$, $s_{os} = 1$. Before independently sharding P , G , and OS , we establish the following constraints:

$$R \geq s_{dp}^0 \geq s_{os}^0 \geq s_g^0 \geq s_p^0, \quad N \geq s_{dp}^1 \geq s_{os}^1 \geq s_g^1 \geq s_p^1,$$

where s_{dp}^0 and s_{dp}^1 is the device mesh of DP ranks, R denotes the GPU count per node, and N is the node number. As shown in Figure 4, adhering to the dependency avoids unnecessary data movement and storage.

C. Communication Time Analysis

In this subsection, we analyze the communication cost associated with individually partitioning parameters, gradients, and optimizer states. Figure 5 provides an overview of the inserted collective communication operations for each component.

1) *Parameters Sharding*: When $s_p^0 \times s_p^1 = s_p > 1$, the Φ parameters of a model are split into s_p shards, with each GPU managing one shard. As shown in Figure 5(a), during each forward and backward pass of every micro-batch in a step, the training system orchestrates the collection of parameters shards from other GPUs to reconstruct the complete set of model weights required for computations. This is achieved using AllGather on s_p GPUs. In each micro-batch of a step, after the gradients are generated during the backward phase, the training framework launches ReduceScatter to aggregate and distribute gradients across s_p GPUs. The training framework performs AllGather and ReduceScatter

at the granularity of a module within a Transformer layer. The input size for i -th module of a layer is $2\Phi_i$ (using FP16). The communication time attributable to parameters sharding for M micro-batches of a step is given by:

$$T_p = ML \sum_{i=0}^K (2t(\text{AG}, 2\Phi_i, s_p^0 \times s_p^1) + t(\text{RS}, 2\Phi_i, s_p^0 \times s_p^1)),$$

where L denotes the number of layers and K is the number of modules of a layer.

Taking Figure 4 (a) as an example, when $s_p^0 \times s_p^1 = 2 \times 1$, AllGather and ReduceScatter are executed within the same node, parameters sharding allows for overlapped communication with computation. During the forward or backward computation of a module, it is feasible to execute AllGather and ReduceScatter for the subsequent module.

2) *Optimizer States Sharding*: When $s_{os}^0 \times s_{os}^1 = s_{os} > 1$, a total of s_{os} GPUs collectively possess a complete duplicate of optimizer states. Following parameters sharding with s_p , each parameters is stored and replicated across s_{dp}/s_p GPUs. In this configuration, optimizer states may exhibit redundancy, with s_{dp}/s_p replicas distributed across the cluster. To reduce this redundancy, we introduce a solution by allowing $s_{os} > s_p$, affording flexibility to reduce redundancy. In this scenario, the optimizer states for Φ/s_p parameters are shared by s_{os}/s_p GPUs, forming an optimizer states sharding subgroup. Illustrated in Figure 4 (b), GPU-0 and GPU-2 share common parameters shards but maintain distinct optimizer states shards, forming an optimizer states sharding subgroup.

After the backward pass of the last micro-batch in a step, each GPU updates Φ/s_{os} parameters based on the optimizer states. Before the update phase, each GPU should gather and aggregate gradients for parameters within its optimizer states. To optimize this process, we employ AllReduce on gradients across s_{dp}/s_p GPUs sharing the same set of parameters (in the amount of Φ/s_p), as illustrated in Figure 5 (c). In Figure 4 (b), we execute AllReduce on GPU-0/2/4/6. Given that $s_{os} > s_p$, each GPU receives additional gradients not managed by its optimizer states. To resolve this issue, we employ a *select & drop* mechanism, enabling each GPU to exclusively select the necessary gradients from the output of the AllReduce. The AllReduce communication time attributed to optimizer states sharding for a given step is expressed as:

$$T_{os}^0 = \frac{2\Phi}{Us_p} \left(t(\text{AR}, U, \frac{s_{dp}^0}{s_p^0} \times \frac{s_{dp}^1}{s_p^1}) \right).$$

Following the completion of parameter updates, it is essential to spread updated parameters among GPUs within the same optimizer states sharding subgroup. For example, In Figure 4 (b), GPU-0 should send its updated parameters to GPU-2. For disseminating updated parameters, the training system utilizes a group of Broadcast operations. Each Broadcast operation processes an input of size $2\Phi/s_{os}$ on average, and the system executes s_{os}/s_p such operations. The Broadcast

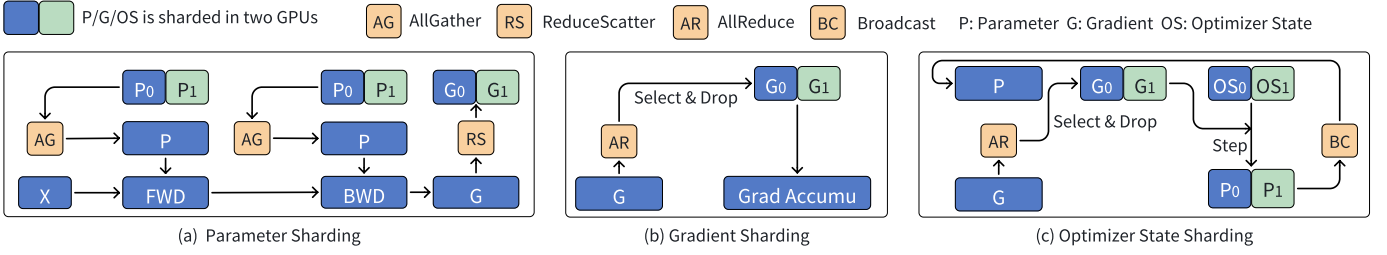


Fig. 5. Analysis of inserted collective communication operations when individually sharding parameters, gradients, and optimizer states.

communication time attributable to optimizer states sharding during a step is given by:

$$T_{os}^1 = \frac{s_{os}}{s_p} \left(t(\text{BC}, \frac{2\Phi}{s_{os}}, \frac{s_{os}^0}{s_p^0} \times \frac{s_{os}^1}{s_p^1}) \right).$$

optimizer states sharding facilitates the potential for overlapped communication and computation. During the backward computation of the i -th layer, we can perform `AllReduce` on gradients generated on layer $i + 1$. Simultaneously, during the forward computation of the i -th layer, it is also possible to broadcast the latest parameters (updated in the previous step) for the next layer.

3) *Gradients Sharding*: When $s_g^0 \times s_g^1 = s_g > 1$, a total of s_g GPUs collectively hold a complete copy of gradients generated at each micro-batch of every step. As depicted in Figure 4, if $s_g = s_p$, each GPU retains Φ/s_p gradients, accumulating them at every micro-batch based on the parameter sharding mechanism. In this work, we introduce the flexibility to set $s_g > s_p$ to conserve GPU memory. For simplicity, we impose the following constraints on the selection of s_g :

$$s_g \in \{s_p, s_{os}\}.$$

In the scenario where $s_g > s_p$, each GPU initiates an `AllReduce` operation on s_g/s_p GPUs to aggregate and distribute gradients in every micro-batch, excluding the last one. Following the aggregation, each GPU retains only the gradients allocated to it, discarding the surplus. For instance, in Figure 4 (c), GPU-0 and GPU-2 can employ such a *select & drop* mechanism to shard gradients. Assuming `AllReduce` is executed with bucket size U , the communication time attributable to gradients sharding of a step can be expressed as:

$$T_g = (M - 1) \frac{2\Phi}{Us_p} \left(t(\text{AR}, U, \frac{s_{dp}^0}{s_g^0} \times \frac{s_{dp}^1}{s_g^1}) \right).$$

Gradients sharding can also overlap communication with computation. During the backward computation for i -th layer, we can concurrently execute `AllReduce` for $(i + 1)$ -th layer.

4) *Summary*: Based on the aforementioned analysis, we can conclude that the single-step communication time of distributed LLM training with a flexible model states sharding strategy is the sum of three components:

$$T_{comm}(s_p^0, s_p^1, s_g^0, s_g^1, s_{os}^0, s_{os}^1) = T_p + T_g + T_{os}^0 + T_{os}^1.$$

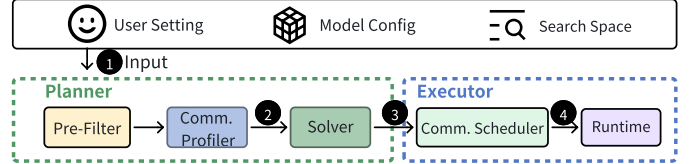


Fig. 6. Overview of `Lins` architecture and workflow. The Planner identifies the optimal solution for model states sharding. The Executor executes LLM training using the selected strategy, and enhances communication performance through overlap and placement optimization.

D. GPU Memory Consumption Analysis

In the context of mixed-precision training with the Adam optimizer, the GPU memory consumed by model states during training can be expressed as the sum of memory allocated for sharded parameters, gradients, and optimization states:

$$D_{modelstate}(s_p^0, s_p^1, s_g^0, s_g^1, s_{os}^0, s_{os}^1) = \frac{2\Phi}{s_p^0 s_p^1} + \frac{2\Phi}{s_g^0 s_g^1} + \frac{12\Phi}{s_{os}^0 s_{os}^1}.$$

Additionally, the total GPU memory consumption, D_{total} , can be encapsulated by:

$$D_{total} = D_{modelstate} + D_{activation} + D_{tmp},$$

where $D_{activation}$ is the memory consumed by activations during training, and D_{tmp} denotes the temporary memory used by communication buffers or other transient variables. Existing methodologies [5] [15] for analyzing and predicting activation memory usage are seamlessly integrated into our present study.

V. SYSTEM DESIGN & COMMUNICATION OVERLAP

To reduce the communication overhead of `ZeRO` for efficient LLM training, we introduce `Lins`. It leverages an expanded model states sharding space and is adept at identifying the most communication-efficient factors. We focus on how `Lins` systematically optimizes the training performance with a flexible model states sharding strategy.

A. System Architecture

Figure 6 illustrates the two components of `Lins`: the Planner and the Executor. (1) The Planner identifies the optimal solution for model states sharding. This component integrates three modules: the Pre-Filter, narrowing the search space based on specific rules; the Communication-Profiler, offering predictions for collective communication time; and the Solver, constructed by a memory and communication cost model to

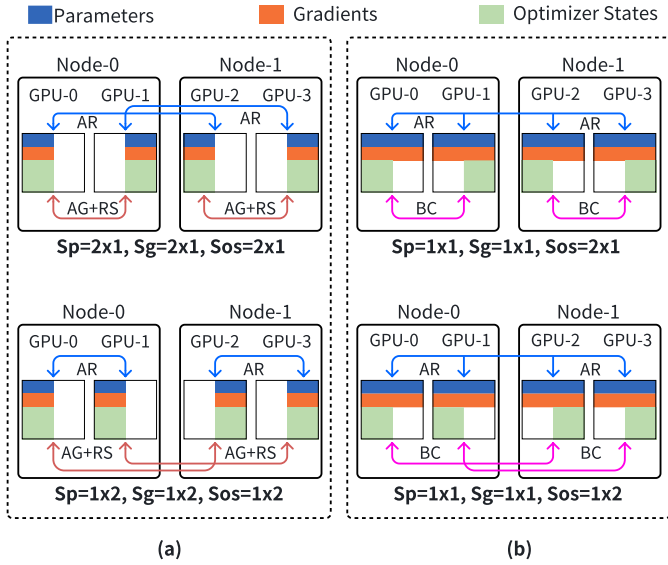


Fig. 7. Example of sharding model states within the same node. In (a) assigning $s_p^0 = 1, s_p^1 = 2$ results in increased cross-node communication caused by AllGather and ReduceScatter. In (b) with $s_{os} = 2$, setting $s_{os}^0 = 1, s_{os}^1 = 2$ creates cross-node Broadcast.

identify the optimal strategy. (2) The Executor is accountable for executing LLM training using the selected strategy. Moreover, Lins employs the topology-aware Communication Placement strategy to reduce network communication across spine switches, enhancing overall efficiency.

Workflow. ① Lins begins by having users define LLM architecture, specifying metadata such as layer number and sequence length, along with hyper-parameters like micro-batch size and micro-batch number. Users also provide settings for the training cluster, including the total number of GPUs and GPU memory capacity. ② The Planner eliminates certain strategies that may incur additional communication costs, resulting in a set of alternative strategies. The Communication-Profiler, operating offline, provides communication time data, aiding the Planner in estimating step time for these alternatives. ③ Using an optimization problem solver, the Planner identifies the optimal strategy. ④ Subsequently, the Executor runs the training job using the chosen strategy, enhanced by a topology-aware Communication Placement strategy.

Lins utilizes real-system profiling to ascertain the effective bandwidth of three used collective communication operations (i.e., AllGather, ReduceScatter, AllReduce and Broadcast) across diverse communication sizes and device meshes. Consequently, Lins estimates the communication latency induced by model states sharding.

B. Execution Planner

The Execution Planner generates the optimal combination strategy for the input model with the provided hardware information. Lins formulates an optimization problem to search for the optimal $\{s_p^0, s_p^1, s_g^0, s_g^1, s_{os}^0, s_{os}^1\}$ by minimizing the sum of communication costs subject to memory constraints. The integer programming problem is defined as follows:

TABLE III
STRATEGIES USED FOR SHARDING P, G AND OS .

Model	Approach ¹	s_p^0	s_p^1	s_g^0	s_g^1	s_{os}^0	s_{os}^1
LLaMA-7B	Our Work	1	1	1	1	8	1
	ZeRO-1	1	1	1	1	R	N
	ZeRO-3	R	N	R	N	R	N
	MiCS	8	1	8	1	8	1
	ZeRO++ ²	R	N	R	N	R	N
LLaMA-13B	Our Work	4	1	4	1	8	1
	ZeRO-3	R	N	R	N	R	N
	MiCS	8	1	8	1	8	1
	ZeRO++ ²	R	N	R	N	R	N
	LLaMA-30B	Our Work	8	1	8	1	8
ZeRO-3		R	N	R	N	R	N
MiCS		8	2	8	2	8	2
ZeRO++ ²		R	N	R	N	R	N

¹ We set $s_{dp}^0 = R, s_{dp}^1 = N$ in this experiment.

² ZeRO++ uses $s_p^0 = 8, s_p^1 = 1$ to shard secondary parameters.

$$\text{Minimize } T_{comm}(s_p^0, s_p^1, s_g^0, s_g^1, s_{os}^0, s_{os}^1) \quad (1)$$

$$\text{Subject to } D_{total} \leq \text{GPU_Memory_Capacity} \quad (2)$$

$$1 \leq s_p^0 \leq s_g^0 \leq s_{os}^0 \leq s_{dp}^0 \leq R \quad (3)$$

$$1 \leq s_p^1 \leq s_g^1 \leq s_{os}^1 \leq s_{dp}^1 \leq N \quad (4)$$

$$s_i^0 \times k = s_{dp}^0, k \in \mathbb{Z}, i \in \{p, g, os\} \quad (5)$$

$$s_j^1 \times k = s_{dp}^1, k \in \mathbb{Z}, j \in \{p, g, os\} \quad (6)$$

$$s_i^0 = s_{dp}^0, \text{ if } s_i^1 > 1, i \in \{p, g, os\} \quad (7)$$

This problem minimizes the communication cost of LLM training with respect to the GPU memory capacity (Equation 2) and the dependency rules outlined in Section IV-B (Equation 3, 4). Instead of exhaustively iterating through all possible solutions for $\{s_p^0, s_p^1, s_g^0, s_g^1, s_{os}^0, s_{os}^1\}$, we optimize the efficiency of assignment strategy exploration by incorporating two filters. Firstly, s_{dp}^0 should be divisible by $s_{p,g,os}^0$ (Equation 5), ensuring the participation of all GPUs within a node in the training process. Additionally, s_{dp}^1 should be divisible by $s_{p,g,os}^1$ (Equation 6), allowing for the utilization of all nodes in the training. Secondly, to minimize cross-node communication (Equation 7), a priority is placed on employing fewer nodes when sharding P, G , and OS independently. For instance, in Figure 7(a), selecting $(s_p^0 = 1, s_p^1 = 2)$ necessitates launching AllGather and ReduceScatter on two nodes every micro-batch, while opting for $(s_p^0 = 2, s_p^1 = 1)$ results in reduced cross-node communication costs. In Figure 7(b), setting $(s_{os}^0 = 1, s_{os}^1 = 2)$ induces cross-node AllGather for spreading updated parameters per step, whereas $(s_{os}^0 = 2, s_{os}^1 = 1)$ confines this communication within a node. Based on these filters, Lins can efficiently employ a brute-force search method to obtain the optimal solution, effectively navigating the solution space with reduced complexity.

VI. EVALUATION

A. Experimental Setup

1) *Implementation:* We use an iterative solver to dynamically optimize communication costs based on the provided

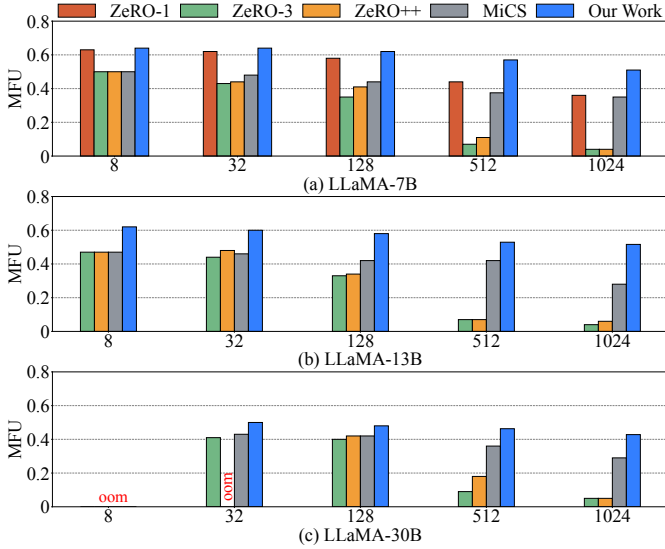


Fig. 8. End-to-end evaluation results (MFU) of model training.

configuration. To uphold comparable computational performance, `Lins` incorporates FlashAttention-v2 [24] and adopts mixed-precision training with BF16, aligning with baseline systems. We also introduce a user-friendly interface enabling users to customize the sharding of P , G , and OS through predefined configurations or leverage the integrated solver to automatically determine the optimal sharding strategy. `Lins` uses specific hooks of PyTorch 2.1, to facilitate the necessary NCCL communications for sharding P , G , and OS . Timely initiation of these operations is important for ensuring both correctness and efficiency.

2) *Testbed*: We evaluate the training performance of three popular LLMs: LLaMA-7B, LLaMA-13B, and LLaMA-30B. The training is conducted on a dedicated cluster with 128 GPU servers. Each server is equipped with 8 GPUs and 128 CPU cores, resulting in a total of 1024 NVIDIA Ampere GPUs (A800). Each GPU is outfitted with 80GB of memory, interconnected through NVLink within a node, and inter-node communication is facilitated by 4 Mellanox HDR InfiniBand without SHARP.

3) *Baselines & Evaluation Metrics*: We conduct a comprehensive benchmark of `Lins`, comparing it against DeepSpeed-ZeRO1, DeepSpeed-ZeRO3 [15], DeepSpeed-ZeRO++ [6], and DeepSpeed-MiCS [7]. Our evaluation focuses on key performance metrics, including Model FLOPs Utilization (MFU)² [25] and Tokens per GPU per Second (TGS). The sequence length is held constant at 4096 tokens in all experiments. The sequence length is fixed at 4096 tokens. Micro-batch size is configured to 1 sequence with 4096 tokens, while the global-

²We calculate FLOPs and MFU using the formula in Megatron-LM. As detailed in [24], while the FLOPs due to attention should be halved, with causal mask, only approximately half the number of elements in attention needs computation. For consistency, we adhere to the literature formula (without dividing attention FLOPs by 2) as in FlashAttention and many other libraries.

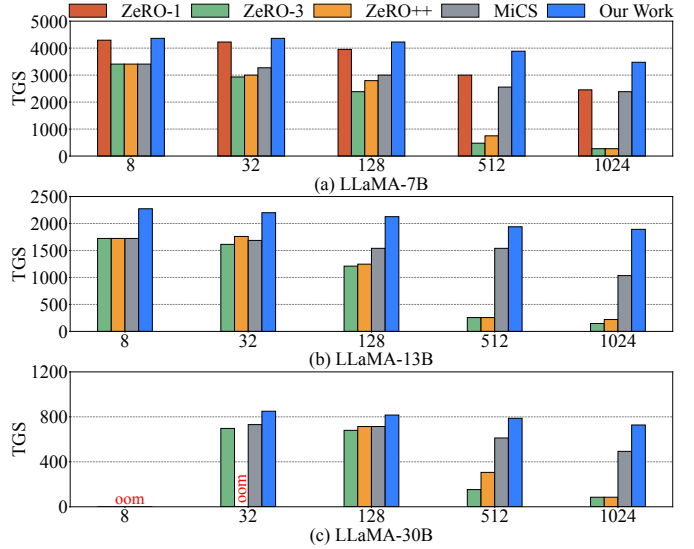


Fig. 9. End-to-end evaluation results (TGS) of model training.

batch size is set to 4 million tokens. The micro-batch number is 128 with 8 GPUs (i.e. $M = 128$); however, training with 1024 GPUs reduces the micro-batch number to 1 (i.e. $M = 1$). Since the core objective of `Lins` is reducing communication overhead in ZeRO, we adopt $s_{dp}^0 = R$, $s_{dp}^1 = N$ across all experiments, excluding tensor or pipeline parallelism.

4) *System Configurations*: Table III presents the configurations utilized in `Lins` and the baselines. `Lins` maintains a uniform set of configurations when scaling training from 8 GPUs to 1024 GPUs. In ZeRO++, the secondary shard number of the parameters is tuned for optimal performance, with ($s_p^0 = 8$, $s_p^1 = 1$), and quantization is not enabled. Activation recomputation is applied during the training of LLaMA-30B, while it is disabled for LLaMA-7B and LLaMA-13B. The communication-computation overlap configurations are consistently enabled in all baselines. To ensure a fair comparison with baselines, we disabled the overlap between Broadcast and forward computation during the end-to-end system evaluations, as these baselines do not provide this function.

B. End-to-End System Evaluation

1) *Scalability Performance*: Figure 8 illustrates the MFU during the training of models of varying sizes with different GPU number, while Figure 9 provides corresponding TGS results. `Lins` exhibits higher performance across all cases than the baseline systems. Specifically, it achieves 51%, 52%, and 42% MFU when training LLaMA-7B, LLaMA-13B, and LLaMA-30B models with 1024 GPUs, respectively.

When training LLaMA-7B with 8 GPUs, ZeRO-1 exhibits a very similar MFU to `Lins`. The observation indicates that both systems achieve comparable computation efficiency. This similarity arises from the fact that they share the same communication cost. Importantly, this result underscores that `Lins`, despite introducing innovative communication optimizations,

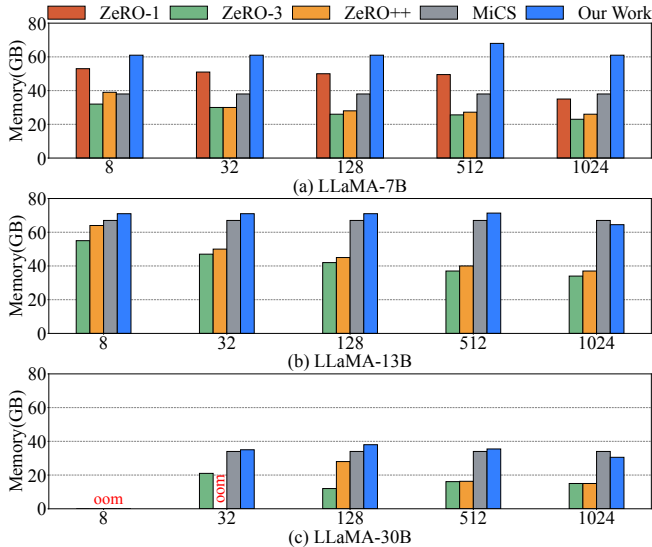


Fig. 10. Peak Memory of training LLaMA-7B/13/30B from 8 to 1024 GPUs.

maintains a comparable level of computation efficiency with baseline systems, given the commonality in utilizing the same computation engine, such as FlashAttention.

As the GPU number increases, *Lins* demonstrates a comparatively stable decrease in MFU and TGS across LLaMA-7B, LLaMA-13B and LLaMA-30B models when compared to other baselines. Expanding from 8 to 1024 GPUs, *Lins* experiences a modest 15% reduction in MFU, while ZeRO-3 can exhibit reductions of up to 88%. The decrease in *Lins*'s MFU is attributed to the reduced computational load per GPU as the number of GPUs grows, leading to a higher communication-to-compute ratio. Notably, the MiCS approach, which employs a subgroup communication strategy, also exhibits a relatively gentle downward trend, similar to *Lins*. However, due to its limited array of configuration options, MiCS consistently maintains an MFU below that of *Lins*. ZeRO++, while often outperforming ZeRO-3, faces challenges such as out-of-memory (OOM) issues. For instance, when training LLaMA-30B on 32 GPUs, ZeRO++ encountered an OOM situation, whereas other methods achieved an MFU of around 40%.

When training LLaMA-7B on 1024 GPUs, *Lins* achieves MFU 51%, surpassing other baselines. ZeRO-1 follows closely with 36% MFU, while MiCS ranks third at 35%. ZeRO-3 and ZeRO++ lag significantly behind, achieving approximately 4% MFU. In comparison to ZeRO-1, *Lins* effectively constrains the Broadcast operation, which is used to disseminate updated parameters to other GPUs at the end of each step, involving only 8 GPUs. This strategic approach minimizes communication overhead. On the other hand, MiCS also reduces the communication scale of AllGather and ReduceScatter within a node for parameters fetch and gradients distribution, yet it generates more traffic than *Lins*. Consequently, MiCS exhibits lower performer compared to ZeRO-1 when training LLaMA-7B on 1024 GPUs. For LLaMA-13B and LLaMA-

30B training on 1024 GPUs, *Lins* maintains its leading position with MFU values of 51% and 43%, respectively. MiCS follows with 33% for LLaMA-13B and 29% for LLaMA-30B.

2) *GPU Memory Analysis*: Figure 10 illustrates the maximum allocated memory during training for various systems. ZeRO-3 stands out as the most memory-efficient, primarily due to its aggressive splitting of model states across all GPUs. The memory consumption of ZeRO-3 becomes stable in large-scale training. Both ZeRO++ and MiCS demonstrate enhanced training performance at the expense of higher memory consumption. MiCS, in particular, prioritizes redundant storage to optimize communication efficiency, resulting in approximately double the memory usage compared to ZeRO-3 for the same models on 1024 GPUs. *Lins* attains higher training efficiency by dynamically managing the memory allocation of parameters, gradients and optimizer states.

VII. RELATED WORK

Model parallelism. Model parallelism is represented by two approaches: tensor parallelism and pipeline parallelism. Tensor parallelism [5] involves partitioning specific layer weights and introducing additional AllReduce communication. Pipeline parallelism [12], [13], [26], [27] divides the layers of the model horizontally among GPUs. Recent innovations have proposed methods that autonomously discern parallelism approaches by intricately melding both data and model parallelism for distinct operators within the model. To illustrate, solutions like Alpa [28], FlexFlow [29], [30], and TensorOpt [31] incorporate both data and tensor parallelism.

Large-scale communication optimization. Some works [4], [17], [18], [32] try to overlap communication with computation to mitigate communication costs. ZeRO++ and Espresso [33] utilize quantization and compression techniques to reduce communication volume, albeit at the expense of precision. DEAR [34] aggregates multiple small communications using fixed-size buffers to reduce communication overheads. Hetu [35] leverages hierarchical all-to-all to minimize inter-node communication volume under poor inter-node communication. Similarly, Hybrid AllReduce [36] attempts to decompose a single collective communication primitive into a combination of multiple subgroup communications.

VIII. CONCLUSION

We propose *Lins* to address the communication challenge of distributed LLM training at scale with ZeRO. The proposed *Lins* introduces a novel approach by incorporating flexible sharding strategies—*Full-Replica*, *Full-Sharding*, and *Partial-Sharding*—for each component within the model states (Parameters, Gradients, and Optimizer States). The introduced sharding factors ($s_p^0 \times s_p^1$, $s_g^0 \times s_g^1$, $s_{os}^0 \times s_{os}^1$) control GPU and device mesh sharding. Analyzing memory and communication costs for each dimension, *Lins* formulates an optimization problem to find factors optimizing communication costs under memory constraints. Compared to MiCS and ZeRO++, *Lins* improves the training throughput by $1.4 \sim 12.7 \times$.

ACKNOWLEDGMENTS

The research is supported under the National Key R&D Program of China (2022ZD0160201) and the RIE2020 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contributions from the industry partner(s).

REFERENCES

- [1] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark *et al.*, “Training compute-optimal large language models,” *arXiv preprint arXiv:2203.15556*, 2022.
- [2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.
- [3] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.
- [4] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, “Pytorch fsdp: Experiences on scaling fully sharded data parallel,” *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3848–3860, 2023.
- [5] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, “Efficient large-scale language model training on gpu clusters using megatron-lm,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
- [6] G. Wang, H. Qin, S. A. Jacobs, C. Holmes, S. Rajbhandari, O. Ruwase, F. Yan, L. Yang, and Y. He, “Zero++: Extremely efficient collective communication for giant model training,” *arXiv preprint arXiv:2306.10209*, 2023.
- [7] Z. Zhang, S. Zheng, Y. Wang, J. Chiu, G. Karypis, T. Chilimbi, M. Li, and X. Jin, “Mics: near-linear scaling for training gigantic model on public cloud,” *Proceedings of the VLDB Endowment*, vol. 16, p. 37–50, 2022.
- [8] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo *et al.*, “Characterization of large language model development in the datacenter,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI’24)*, 2024.
- [9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” vol. 30, 2017.
- [11] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, “Pytorch distributed: Experiences on accelerating data parallel training,” *arXiv preprint arXiv:2006.15704*, 2020.
- [12] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” vol. 32, 2019.
- [13] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, “Pipedream: Generalized pipeline parallelism for dnn training,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.
- [14] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.
- [15] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “Zero: Memory optimizations toward training trillion parameter models,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.
- [16] S. Li, H. Liu, Z. Bian, J. Fang, H. Huang, Y. Liu, B. Wang, and Y. You, “Colossal-ai: A unified deep learning system for large-scale parallel training,” in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 766–775.
- [17] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, “A generic communication scheduler for distributed dnn training acceleration,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.
- [18] P. Sun, Y. Wen, R. Han, W. Feng, and S. Yan, “Gradientflow: Optimizing network performance for large-scale distributed dnn training,” *IEEE Transactions on Big Data*, vol. 8, no. 2, pp. 495–507, 2019.
- [19] P. Sun, Y. Wen, T. N. B. Duong, and S. Yan, “Timed dataflow: Reducing communication overhead for distributed machine learning systems,” in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2016, pp. 1110–1117.
- [20] R. Thakur and W. D. Gropp, “Improving the performance of collective operations in mpich,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer, 2003, pp. 257–267.
- [21] P. Sanders, J. Speck, and J. L. Träff, “Two-tree algorithms for full bandwidth broadcast, reduction and scan,” *Parallel Computing*, vol. 35, no. 12, pp. 581–594, 2009.
- [22] Z. Li, J. Huang, Y. Li, A. Xu, S. Zhou, J. Liu, and J. Wang, “A2tp: Aggregator-aware in-network aggregation for multi-tenant learning,” in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 639–653.
- [23] S. Liu, Q. Wang, J. Zhang, W. Wu, Q. Lin, Y. Liu, M. Xu, M. Canini, R. C. Cheung, and J. He, “In-network aggregation with transport transparency for distributed training,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 376–391.
- [24] T. Dao, “Flashattention-2: Faster attention with better parallelism and work partitioning,” *arXiv preprint arXiv:2307.08691*, 2023.
- [25] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, “Palm: Scaling language modeling with pathways,” *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.
- [26] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, “Dapple: A pipelined data parallel approach for training large models,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.
- [27] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, “Pipemare: Asynchronous pipeline parallel dnn training,” *Proceedings of Machine Learning and Systems*, vol. 3, pp. 269–296, 2021.
- [28] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing *et al.*, “Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 559–578.
- [29] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.
- [30] C. Unger, Z. Jia, W. Wu, S. Lin, M. Baines, C. E. Q. Narvaez, V. Ramakrishnaiah, N. Prajapati, P. McCormick, J. Mohd-Yusof *et al.*, “Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization,” in *16th USENIX Symposium on Operating Systems Design and Implementation*, 2022, pp. 267–284.
- [31] Z. Cai, X. Yan, K. Ma, Y. Wu, Y. Huang, J. Cheng, T. Su, and F. Yu, “Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1967–1981, 2021.
- [32] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, “Priority-based parameter propagation for distributed dnn training,” *Proceedings of Machine Learning and Systems*, vol. 1, pp. 132–145, 2019.
- [33] Z. Wang, H. Lin, Y. Zhu, and T. E. Ng, “Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies,” in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 867–882.
- [34] L. Zhang, S. Shi, X. Chu, W. Wang, B. Li, and C. Liu, “Dear: accelerating distributed deep learning with fine-grained all-reduce pipelining,” in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2023, pp. 142–153.
- [35] X. Nie, P. Zhao, X. Miao, T. Zhao, and B. Cui, “Hetumoe: An efficient trillion-scale mixture-of-expert distributed training system,” *arXiv preprint arXiv:2203.14685*, 2022.
- [36] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters,” pp. 3505–3506, 2020.